2ND EDITION

THE IDA PRO BOOK

THE *UNOFFICIAL GUIDE* TO THE WORLD'S MOST POPULAR DISASSEMBLER

CHRIS EAGLE

"I wholeheartedly recommend The IDA Pro Book *to all IDA Pro users."*
—Ilfak Guilfanov, creator of IDA Pro

no starch press

THE IDA PRO BOOK

2ND EDITION

The Unofficial Guide to the World's Most Popular Disassembler

by Chris Eagle

no starch press

San Francisco

BRIEF CONTENTS

PART IV: EXTENDING IDA'S CAPABILITIES

PART V: REAL-WORLD APPLICATIONS

PART VI: THE IDA DEBUGGER

15

IDA SCRIPTING

It is a simple fact that no application can meet every need of every user. It is just not possible to anticipate every potential use case that may arise. Application developers are faced with the choice of responding to an endless stream of feature requests or offering users a means to solve their own problems. IDA takes the latter approach by integrating scripting features that allow users to exercise a tremendous amount of programmatic control over IDA's actions.

Potential uses for scripts are infinite and can range from simple one-liners to full-blown programs that automate common tasks or perform complex analysis functions. From an automation standpoint, IDA scripts can be viewed as macros,[1] while from an analysis point of view, IDA's scripting languages serve as the query languages that provide programmatic access to the contents of an IDA database. IDA supports scripting using two different

1. Many applications offer facilities that allow users to record sequences of actions into a single complex action called a *macro*. Replaying or triggering a macro causes the entire sequence of recorded steps to be executed. Macros provided an easy means to automate a complex series of actions.

languages. IDA's original, embedded scripting language is named *IDC*, perhaps because its syntax bears a close resemblance to C. Since the release of IDA 5.4,[2] integrated scripting with Python has also been supported through the integration of the IDAPython plug-in by Gergely Erdelyi.[3] For the remainder of this chapter we will cover the basics of writing and executing both IDC and Python scripts as well as some of the more useful functions available to script authors.

Basic Script Execution

Before diving into the details of either scripting language, it is useful to understand the most common ways that scripts can be executed. Three menu options, File ▸ Script File, File ▸ IDC Command, and File ▸ Python Command[4] are available to access IDA's scripting engine. Selecting File ▸ Script File indicates that you wish to run a standalone script, at which point you are presented with a file-selection dialog that lets you choose the script to run. Each time you run a new script, the program is added to a list of recent scripts to provide easy access to edit or rerun the script. Figure 15-1 shows the Recent Scripts window accessible via the View ▸ Recent Scripts menu option.

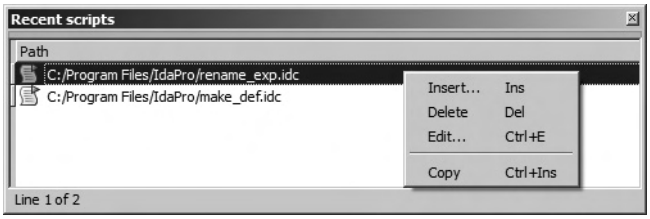

Figure 15-1: The Recent Scripts window

Double-clicking a listed script causes the script to be executed. A pop-up, context-sensitive menu offers options to remove a script from the list or to open a script for editing using the editor specified under Options ▸ General on the Misc tab.

As an alternative to executing a standalone script file, you may elect to open a script entry dialog using File ▸ IDC Command or File ▸ Python Command. Figure 15-2 shows the resulting script entry dialog (for an IDC script in this case), which is useful in situations where you wish to execute only a few statements but don't want to go to the trouble of creating a standalone script file.

2. For a comprehensive list of features introduced with each new version of IDA, visit *http://www.hex-rays.com/idapro/idanew48.htm*.

3. See *http://code.google.com/p/idapython/*.

4. This option is only available if Python is properly installed. Refer to Chapter 3 for details.

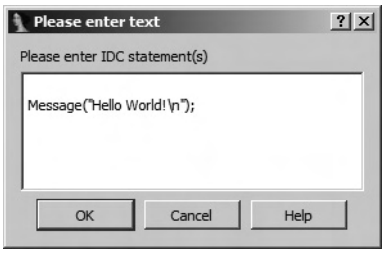

Figure 15-2: The script entry dialog

Some restrictions apply to the types of statements that you can enter in the script dialog, but the dialog is very useful in cases where creating a full-blown script file is overkill.

The last way to easily execute script commands is to use IDA's command line. The command line is available only in GUI versions of IDA, and its presence is controlled by the value of the `DISPLAY_COMMAND_LINE` option in *<IDADIR>/cfg/idagui.cfg*. The command line has been enabled by default since IDA 5.4. Figure 15-3 shows the command line as it appears in the lower-left corner of the IDA workspace, beneath the output window.

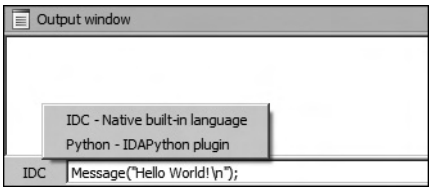

Figure 15-3: The IDA command line

The interpreter that will be used to execute the command line is labeled to the left of the command-line entry box. In Figure 15-3, the command line is configured to execute IDC statements. Clicking this label opens the pop-up menu shown in Figure 15-3, allowing either interpreter (IDC or Python) to be associated with the command line.

Although the command line contains only a single line of text, you can enter multiple statements by separating each statement with a semicolon. As a convenience, the history of recent commands is accessible with the up arrow key. If you find yourself frequently needing to execute very short scripts, you will find the command line very useful.

With a basic ability to execute scripts under our belts, it is time to focus on the specifics of IDA's two available scripting languages, IDC and Python. We begin with a description of IDA's native scripting language, IDC, and conclude with a discussion of IDA's Python integration, which will rely heavily on the foundation built by the IDC sections that follow.

The IDC Language

Unlike for some other aspects of IDA, a reasonable amount of help is available for the IDC language in IDA's help system. Topics available at the top level of the help system include *IDC language*, which covers the basics of IDC syntax, and *Index of IDC functions*, which provides an exhaustive list of built-in functions available to IDC programmers.

IDC is a scripting language that borrows most of its syntactic elements from C. Beginning with IDA 5.6, IDC actually takes on more of the flavor of C++ with the introduction of object-oriented features and exception handling. Because of its similarity to C and C++, we will describe IDC in terms of these languages and focus primarily on where IDC differs.

IDC Variables

IDC is a loosely typed language, meaning that variables have no explicit type. The three primary datatypes used in IDC are integers (IDA documentation uses the type name *long*), strings, and floating point values, with the overwhelming majority of operations taking place on integers and strings. Strings are treated as a native datatype in IDC, and there is no need to keep track of the space required to store a string or whether a string is null terminated or not. Beginning with IDA 5.6, IDC incorporates a number of additional variable types, including objects, references, and function pointers.

All variables must be declared prior to their use. IDC supports local variables and, since IDA 5.4, global variables as well. The IDC keyword `auto` is used to introduce a local variable declaration, and local variable declarations may include initial values. The following examples show legal IDC local variable declarations:

```
auto addr, reg, val;   // legal, multiple variables declared with no initializers
auto count = 0;        // declaration with initialization
```

IDC recognizes C-style multiline comments using `/* */` and C++–style line-terminating comments using `//`. Also, note that several variables may be declared in a single statement and that all statements in IDC are terminated using a semicolon (as in C). IDC does not support C-style arrays (slices are introduced in IDA 5.6), pointers (though references are supported beginning with IDA 5.6), or complex datatypes such as structs and unions. Classes are introduced in IDA 5.6.

Global variable declarations are introduced using the `extern` keyword, and their declarations are legal both inside and outside of any function definition. It is not legal to provide an initial value when a global variable is declared. The following listing shows the declaration of two global variables.

```
extern outsideGlobal;

static main() {
   extern insideGlobal;
   outsideGlobal = "Global";
   insideGlobal = 1;
}
```

Global variables are allocated the first time they are encountered during an IDA session and persist as long as that session remains active, regardless of the number of databases that you may open and close.

IDC Expressions

With a few exceptions, IDC supports virtually all of the arithmetic and logical operators available in C, including the ternary operator (`? :`). Compound assignment operators of the form `op=` (`+=`, `*=`, `>>=`, and the like) are not supported. The comma operator is supported beginning with IDA 5.6. All integer operands are treated as signed values. This affects integer comparisons (which are always signed) and the right-shift operator (`>>`), which always performs an arithmetic shift with sign bit replication. If you require logical right shifts, you must implement them yourself by masking off the top bit of the result, as shown here:

```
result = (x >> 1) & 0x7fffffff;  //set most significant bit to zero
```

Because strings are a native type in IDC, some operations on strings take on a different meaning than they might in C. The assignment of a string operand into a string variable results in a string copy operation; thus there is no need for string copying or duplicating functions such as C's `strcpy` and `strdup`. Also, the addition of two string operands results in the concatenation of the two operands; thus "Hello" + "World" yields "HelloWorld"; there is no need for a concatenation function such as C's `strcat`. Starting with IDA 5.6, IDC offers a slice operator for use with strings. Python programmers will be familiar with slices, which basically allow you to specify subsequences of array-like variables. Slices are specified using square brackets and a start (inclusive) and end (exclusive) index. At least one index is required. The following listing demonstrates the use of IDC slices.

```
auto str = "String to slice";
auto s1, s2, s3, s4;
s1 = str[7:9];     // "to"
s2 = str[:6];      // "String", omitting start index starts at 0
s3 = str[10:];     // "slice", omitting end index goes to end of string
s4 = str[5];       // "g", single element slice, similar to array element access
```

Note that while there are no array datatypes available in IDC, the slice operator effectively allows you to treat IDC strings as if they were arrays.

IDC Statements

As in C, all simple statements are terminated with a semicolon. The only C-style compound statement that IDC does not support is the `switch` statement. When using `for` loops, keep in mind that IDC does not support compound assignment operators, which may affect you if you wish to count by anything other than one, as shown here:

```
auto i;
for (i = 0; i < 10; i += 2) {}     // illegal, += is not supported
for (i = 0; i < 10; i = i + 2) {}  // legal
```

With IDA 5.6, IDC introduces `try/catch` blocks and the associated `throw` statement, which are syntactically similar to C++ exceptions.[5] IDA's built-in help contains specifics on IDC's exception-handling implementation.

For compound statements, IDC utilizes the same bracing ({}) syntax and semantics as C. Within a braced block, it is permissible to declare new variables as long as the variable declarations are the first statements within the block. However, IDC does not rigorously enforce the scope of the newly introduced variables, because such variables may be referenced beyond the block in which they were declared. Consider the following example:

```
if (1) {    //always true
   auto x;
   x = 10;
}
else {      //never executes
   auto y;
   y = 3;
}
Message("x = %d\n", x);   // x remains accessible after its block terminates
Message("y = %d\n", y);   // IDC allows this even though the else did not execute
```

The output statements (the `Message` function is analogous to C's `printf`) will inform us that `x = 10` and `y = 0`. Given that IDC does not strictly enforce the scope of `x`, it is not terribly surprising that we are allowed to print the value of `x`. What is somewhat surprising is that `y` is accessible at all, given that the block in which `y` is declared is never executed. This is simply a quirk of IDC. Note that while IDC may loosely enforce variable scoping within a function, variables declared within one function continue to remain inaccessible in any other function.

IDC Functions

IDC supports user-defined functions in standalone programs (*.idc* files) only. User-defined functions are not supported when using the IDC command dialog (see "Using the IDC Command Dialog" on page 255). IDC's syntax for declaring user-defined functions is where it differs most from C. The `static`

5. See *http://www.cplusplus.com/doc/tutorial/exceptions/*.

keyword is used to introduce a user-defined function, and the function's parameter list consists solely of a comma-separated list of parameter names. The following listing details the basic structure of a user-defined function:

```
static my_func(x, y, z) {
   //declare any local variables first
   auto a, b, c;
   //add statements to define the function's behavior
   // ...
}
```

Prior to IDA 5.6, all function parameters are strictly call-by-value. Call-by-reference parameter passing was introduced with IDA 5.6. Interestingly, whether a parameter is passed using call-by-value or call-by-reference is determined by the manner in which the function is called, not the manner in which the function is declared. The unary `&` operator is used in a function call (*not* the function declaration) to denote that an argument is being passed by reference. The following examples show invocations of the `my_func` function from the previous listing making use of both call-by-value and call-by-reference parameter passing.

```
auto q = 0, r = 1, s = 2;
my_func(q, r, s);   //all three arguments passed using call-by-value
                    //upon return, q, r, and s hold 0, 1, and 2 respectively
my_func(q, &r, s);  //q and s passed call-by-value, r is passed call-by-reference
                    //upon return, q, and s hold 0 and 2 respectively, but r may have
                    //changed. In this second case, any changes that my_func makes to its
                    //formal parameter y will be reflected in the caller as changes to r
```

Function declarations never indicate whether a function explicitly returns a value or what type of value is returned when a function does yield a result.

USING THE IDC COMMAND DIALOG

The IDC command dialog offers a simple interface for entering short sequences of IDC code. The command dialog is a great tool for rapidly entering and testing new scripts without the hassle of creating a standalone script file. The most important thing to keep in mind when using the command dialog is that you *must not* define any functions inside the dialog. In essence, IDA wraps your statements within a function and then calls that function in order to execute your statements. If you were to define a function within the dialog, the net effect would be a function defined within a function, and since nested function declarations are not allowed in IDC (or in C for that matter), a syntax error would result.

When you wish to return a value from a function, use a `return` statement to return the desired value. It is permissible to return entirely different datatypes from different paths of execution within a function. In other words, a function may return a string in some cases, while in other cases the same

function may return an integer. As in C, use of a `return` statement within a function is optional. However, unlike C, any function that does not explicitly return a value implicitly returns the value zero.

As a final note, beginning with IDA 5.6, functions take a step closer to becoming first-class objects in IDC. It is now possible to pass function references as arguments to other functions and return function references as the result of a function. The following listing demonstrates the use of function parameters and functions as return values.

```
static getFunc() {
   return Message;  //return the built-in Message function as a result
}

static useFunc(func, arg) {  //func here is expected to be a function reference
   func(arg);
}

static main() {
   auto f = getFunc();
   f("Hello World\n");       //invoke the returned function f
   useFunc(f, "Print me\n"); //no need for & operator, functions always call-by-reference
}
```

IDC Objects

Another feature introduced in IDA 5.6 is the ability to define classes and, as a result, have variables that represent objects. In the discussion that follows, we assume that you have some familiarity with an object-oriented programming language such as C++ or Java.

IDA SCRIPTING EVOLVES

If you haven't gotten the idea that a large number of changes to IDC were introduced with IDA 5.6, then you haven't been paying attention. Following the integration of IDAPython in IDA 5.4, Hex-Rays looked to rejuvenate IDC, resulting in many of the features mentioned in this chapter being introduced in IDA 5.6. Along the way, JavaScript was even contemplated as a potential addition to IDA's scripting lineup.*

*See *http://www.hexblog.com/?p=101*.

IDC defines a root class named `object` from which all classes ultimately derive, and single inheritance is supported when creating new classes. IDC does not make use of access specifiers such as `public` and `private`; all class members are effectively public. Class declarations contain only the definitions of the class's member functions. In order to create data members within a class, you simply create an assignment statement that assigns a value to the data member. The following listing will help to clarify.

```
class ExampleClass {
   ExampleClass(x, y) {   //constructor
      this.a = x;         //all ExampleClass objects have data member a
      this.b = y;         //all ExampleClass objects have data member b
   }
   ~ExampleClass() {      //destructor
   }
   foo(x) {
      this.a = this.a + x;
   }
   //...   other member functions as desired
};

static main() {
   ExampleClass ex;            //DON'T DO THIS!! This is not a valid variable declaration
   auto ex = ExampleClass(1, 2);   //reference variables are initialized by assigning
                                   //the result of calling the class constructor
   ex.foo(10);                 //dot notation is used to access members
   ex.z = "string";            //object ex now has a member z, BUT the class does not
}
```

For more information on IDC classes and their syntax, refer to the appropriate section within IDA's built-in help file.

IDC Programs

For any scripting applications that require more than a few IDC statements, you are likely to want to create a standalone IDC program file. Among other things, saving your scripts as programs gives you some measure of persistence and portability.

IDC program files require you to make use of user-defined functions. At a minimum, you must define a function named `main` that takes no arguments. In most cases, you will also want to include the file *idc.idc* in order to pick up useful macro definitions that it contains. The following listing details the components of a minimal IDC program file:

```
#include <idc.idc>    // useful include directive
//declare additional functions as required
static main() {
   //do something fun here
}
```

IDC recognizes the following C-style preprocessor directives:

`#include <`*file*`>`
: Includes the named file in the current file.

`#define <`*name*`>` [*optional value*]
: Creates a macro named *name* and optionally assigns it the specified value. IDC predefines a number of macros that may be used to test various aspects of your script's execution environment. These include *_NT_*,

*_LINUX_*, *_MAC_*, *_GUI_*, and *_TXT_* among others. See the *Predefined symbols* section of the IDA help file for more information on these and other symbols.

`#ifdef <name>`
Tests for the existence of the named macro and optionally processes any statements that follow if the named macro exists.

`#else`
Optionally used in conjunction with an `#ifdef` to provide an alternative set of statements to process in the event the named macro does not exist.

`#endif`
This is a required terminator for an `#ifdef` or `#ifdef/#else` block.

`#undef <name>`
Deletes the named macro.

Error Handling in IDC

No one is ever going to praise IDC for its error-reporting capabilities. There are two types of errors that you can expect to encounter when running IDC scripts: parsing errors and runtime errors.

Parsing errors are those errors that prevent your program from ever being executed and include such things as syntax errors, references to undefined variables, and supplying an incorrect number of arguments to a function. During the parsing phase, IDC reports only the first parsing error that it encounters. In some cases, error messages correctly identify both the location and the type of an error (`hello_world.idc,20: Missing semicolon`), while in other cases, error messages offer no real assistance (`Syntax error near: <END>`). Only the first error encountered during parsing is reported. As a result, in a script with 15 syntax errors, it may take 15 attempts at running the script before you are informed of every error.

Runtime errors are generally encountered less frequently than parsing errors. When encountered, runtime errors cause a script to terminate immediately. One example of a runtime error results from an attempt to call an undefined function that for some reason is not detected when the script is initially parsed. Another problem arises with scripts that take an excessive amount of time to execute. Once a script is started, there is no easy way to terminate the script if it inadvertently ends up in an infinite loop or simply takes longer to execute than you are willing to wait. Once a script has executed for more than two to three seconds, IDA displays the dialog shown in Figure 15-4.

Figure 15-4: Script cancellation dialog

This dialog is the only means by which you can terminate a script that fails to terminate properly.

Debugging is another of IDC's weak points. Other than liberal use of output statements, there is no way to debug IDC scripts. With the introduction of exception handling (`try/catch`) in IDA 5.6, it does become possible to build more robust scripts that can terminate or continue as gracefully as you choose.

Persistent Data Storage in IDC

Perhaps you are the curious type who, not trusting that we would provide sufficient coverage of IDA's scripting capability, raced off to see what the IDA help system has to say on the subject. If so, welcome back, and if not, we appreciate you sticking with us this far. In any case, somewhere along the way you may have acquired knowledge that claims that IDC does in fact support arrays, in which case you must surely be questioning the quality of this book. We urge you to give us a chance to sort out this potential confusion.

As mentioned previously, IDC does not support arrays in the traditional sense of declaring a large block of storage and then using a subscript notation to access individual items within that block. However, IDA's documentation on scripting does mention something called *global persistent arrays*. IDC global arrays are better thought of as *persistent named objects*. The objects just happen to be sparse arrays.[6] Global arrays are stored within an IDA database and are persistent across script invocations and IDA sessions. Data is stored in global arrays by specifying an index and a data value to be stored at the specified index in the array. Each element in an array can simultaneously hold one integer value and one string value. IDC's global arrays provide no means for storing floating point values.

NOTE *For the overly curious, IDA's internal mechanism for storing persistent arrays is called a netnode. While the array-manipulation functions described next provide an abstracted interface to netnodes, lower-level access to netnode data is available using the IDA SDK, which is discussed, along with netnodes, in Chapter 16.*

All interaction with global arrays occurs through the use of IDC functions dedicated to array manipulation. Descriptions of these functions follow:

`long CreateArray(string name)`
: This function creates a persistent object with the specified name. The return value is an integer handle required for all future access to the array. If the named object already exists, the return value is –1.

`long GetArrayId(string name)`
: Once an array has been created, subsequent access to the array must be done through an integer handle, which can be obtained by looking up the array name. The return value for this function is an integer handle to be used for all future interaction with the array. If the named array does not exist, the return value is –1.

6. Sparse arrays do not necessarily preallocate space for the entire array, nor are they limited to a particular maximum index. Instead, space for array elements is allocated on an as-needed basis when elements are added to the array.

long SetArrayLong(long id, long idx, long value)
Stores an integer value into the array referred to by id at the position specified by idx. The return value is 1 on success or 0 on failure. The operation will fail if the array id is invalid.

long SetArrayString(long id, long idx, string str)
Stores a string value into the array referred to by id at the position specified by idx. The return value is 1 on success or 0 on failure. The operation will fail if the array id is invalid.

string or long GetArrayElement(long tag, long id, long idx)
While there are distinct functions for storing data into an array depending on the type of data to be stored, there is only one function for retrieving data from an array. This function retrieves either an integer or a string value from the specified index (idx) in the specified array (id). Whether an integer or a string is retrieved is determined by the value of the tag parameter, which must be one of the constants AR_LONG (to retrieve an integer) or AR_STR (to retrieve a string).

long DelArrayElement(long tag, long id, long idx)
Deletes the contents of the specified array location from the specified array. The value of tag determines whether the integer value or string value associated with the specified index is deleted.

void DeleteArray(long id)
Deletes the array referenced by id and all of its associated contents. Once an array has been created, it continues to exist, even after a script terminates, until a call is made to DeleteArray to remove the array from the database in which it was created.

long RenameArray(long id, string newname)
Renames the array referenced by id to newname. Returns 1 if successful or 0 if the operation fails.

Possible uses for global arrays include approximating global variables, approximating complex datatypes, and providing persistent storage across script invocations. Global variables for a script are simulated by creating a global array when the script begins and storing global values in the array. These global values are shared either by passing the array handle to functions requiring access to the values or by requiring any function that requires access to perform a name lookup for the desired array.

Values stored in an IDC global array persist for the lifetime of the database in which the script was executed. You may test for the existence of an array by examining the return value of the CreateArray function. If the values stored in an array are applicable only to a specific invocation of a script, then the array should be deleted before the script terminates. Deleting the array ensures that no global values carry over from one execution of a script to a subsequent execution of the same script.

Associating IDC Scripts with Hotkeys

Occasionally you may develop a script so amazing in its utility that you must have access to it with a keystroke or two. When this happens, you will want to assign a hotkey sequence that you can use to quickly activate your script. Fortunately IDA provides a simple means to do this. Every time IDA is launched, the script contained in *<IDADIR>/idc/ida.idc* is executed. The default version of this script contains an empty `main` function and thus does nothing. To associate a hotkey with one of your scripts, you need to add two lines to *ida.idc*. The first line you must add is an `include` directive to include your script file in *ida.idc*. The second line you must add is a call, within `main`, to the `AddHotkey` function to associate a specific hotkey with your amazing IDC function. This might leave *ida.idc* looking like this:

```
#include <idc.idc>
#include <my_amazing_script.idc>
static main() {
   AddHotkey("z", "MyAmazingFunc");  //Now 'z' invokes MyAmazingFunc
}
```

If the hotkey you are attempting to associate with your script has already been assigned to another IDA action (menu hotkey or plug-in activation sequence), `AddHotkey` silently fails with no way to detect the failure other than the fact that your function fails to execute when your hotkey sequence is activated.

Two important points here are that the standard include directory for IDC scripts is *<IDADIR>/idc* and that you must not name your script function `main`. If you want IDA to find your script easily, you can copy it into *<IDADIR>/idc*. If you intend to leave your script file in another location, then you will need to specify the full path to your script in the `include` statement. While testing your script, it will be useful to run your script as a standalone program with a `main` function. Once you are ready to associate your script with a hotkey, however, you cannot use the name `main`, because it will conflict with the `main` function in *ida.idc*. You must rename your `main` function and use the new name in the call to `AddHotkey`.

Useful IDC Functions

At this point, you have all the information required to write well-formed IDC scripts. What you are lacking is the ability to perform any useful interaction with IDA itself. IDC provides a long list of built-in functions that offer many different ways to access a database. All of the functions are documented to some degree in the IDA help system under the topic *Index of IDC functions*. In most cases, the documentation is nothing more than relevant lines copied from the main IDC include file, *idc.idc*. Becoming comfortable with the rather terse documentation is one of the more frustrating aspects of learning IDC. In general, there is no easy way to answer the question "How do I do *x* in IDC?" The most common way to figure out how to do something is to browse

the list of IDC functions looking for one that, based on its name, appears to do what you need. This presumes, of course, that the functions are named according to their purpose, but their purpose may not always be obvious. For example, in many cases, functions that retrieve information from the database are named `Get`*`XXX`*; however; in many other cases, the `Get` prefix is not used. Functions that change the database may be named `Set`*`XXX`*, `Make`*`XXX`*, or something else entirely. In summary, if you want to use IDC, get used to browsing the list of functions and reading through their descriptions. If you find yourself at a complete loss, don't be afraid to use the support forums at Hex-Rays.[7]

The intent of the remainder of this section is to point out some of the more useful (in our experience) IDC functions and group them into functional areas. Even if you intend to script in Python only, familiarity with the listed functions will be useful to you because IDAPython provides Python equivalents to each function listed here. We make no attempt to cover every IDC function, however, since they are already covered in the IDA help system.

Functions for Reading and Modifying Data

The following functions provide access to individual bytes, words, and double words in a database:

`long Byte(long addr)`
Reads a byte value from virtual address `addr`.

`long Word(long addr)`
Reads a word (2-byte) value from virtual address `addr`.

`long Dword(long addr)`
Reads a double word (4-byte) value from virtual address `addr`.

`void PatchByte(long addr, long val)`
Sets a byte value at virtual address `addr`.

`void PatchWord(long addr, long val)`
Sets a word value at virtual address `addr`.

`void PatchDword(long addr, long val)`
Sets a double word value at virtual address `addr`.

`bool isLoaded(long addr)`
Returns 1 if `addr` contains valid data, 0 otherwise.

Each of these functions takes the byte ordering (little-endian or big-endian) of the current processor module into account when reading and writing the database. The `Patch`*`XXX`* functions also trim the supplied value to an appropriate size by using only the proper number of low-order bytes according to the function called. For example, a call to `PatchByte(0x401010, 0x1234)` will patch location `0x401010` with the byte value `0x34` (the low-order byte of `0x1234`). If an invalid address is supplied while reading the database with `Byte`, `Word`, and `Dword`, the values `0xFF`, `0xFFFF`, and `0xFFFFFFFF` will be returned, respectively. Because there is no way to distinguish these error

7. The support forum is currently located at *http://www.hex-rays.com/forum/*.

values from legitimate data stored in the database, you may wish to call `isLoaded` to determine whether an address in the database contains any data prior to attempting to read from that address.

Because of a quirk in refreshing IDA's disassembly view, you may find that the results of a patch operation are not immediately visible. In such cases, scrolling away from the patched location and then scrolling back to the patched location generally forces the display to be updated properly.

User Interaction Functions

In order to perform any user interaction at all, you will need to familiarize yourself with IDC input/output functions. The following list summarizes some of IDC's more useful interface functions:

`void Message(string format, ...)`
: Prints a formatted message to the output window. This function is analogous to C's `printf` function and accepts a `printf`-style format string.

`void print(...)`
: Prints the string representation of each argument to the output window.

`void Warning(string format, ...)`
: Displays a formatted message in a dialog.

`string AskStr(string default, string prompt)`
: Displays an input dialog asking the user to enter a string value. Returns the user's string or 0 if the dialog was canceled.

`string AskFile(long doSave, string mask, string prompt)`
: Displays a file-selection dialog to simplify the task of choosing a file. New files may be created for saving data (`doSave = 1`), or existing files may be chosen for reading data (`doSave = 0`). The displayed list of files may be filtered according to `mask` (such as `*.*` or `*.idc`). Returns the name of the selected file or 0 if the dialog was canceled.

`long AskYN(long default, string prompt)`
: Prompts the user with a yes or no question, highlighting a default answer (1 = yes, 0 = no, –1 = cancel). Returns an integer representing the selected answer.

`long ScreenEA()`
: Returns the virtual address of the current cursor location.

`bool Jump(long addr)`
: Jumps the disassembly window to the specified address.

Because IDC lacks any debugging facilities, you may find yourself using the `Message` function as your primary debugging tool. Several other `Ask`*`XXX`* functions exist to handle more specialized input cases such as integer input. Please refer to the help system documentation for a complete list of available `Ask`*`XXX`* functions. The `ScreenEA` function is very useful for picking up the current cursor location when you wish to create a script that tailors its behavior

based on the location of the cursor. Similarly, the `Jump` function is useful when you have a script that needs to call the user's attention to a specific location within the disassembly.

String-Manipulation Functions

Although simple string assignment and concatenation are taken care of with basic operators in IDC, more complex operations must be performed using available string-handling functions, some of which are detailed here:

`string form(string format, ...)  // pre IDA 5.6`
Returns a new string formatted according to the supplied format strings and values. This is the rough equivalent to C's `sprintf` function.

`string sprintf(string format, ...)  // IDA 5.6+`
With IDA 5.6, `sprintf` replaces `form` (see above).

`long atol(string val)`
Converts the decimal value `val` to its corresponding integer representation.

`long xtol(string val)`
Converts the hexadecimal value `val` (which may optionally begin with `0x`) to its corresponding integer representation.

`string ltoa(long val, long radix)`
Returns a string representation of `val` in the specified `radix` (2, 8, 10, or 16).

`long ord(string ch)`
Returns the ASCII value of the one-character string `ch`.

`long strlen(string str)`
Returns the length of the provided string.

`long strstr(string str, string substr)`
Returns the index of `substr` within `str` or –1 if the substring is not found.

`string substr(string str, long start, long end)`
Returns the substring containing the characters from `start` through `end-1` of `str`. Using slices (IDA 5.6+) this function is equivalent to `str[start:end]`.

Recall that there is no character datatype in IDC, nor is there any array syntax. Lacking slices, if you want to iterate through the individual characters within a string, you must take successive one-character substrings for each character in the string.

File Input/Output Functions

The output window may not always be the ideal place to send the output of your scripts. For scripts that generate a large amount of text or scripts that generate binary data, you may wish to output to disk files instead. We have

already discussed using the `AskFile` function to ask a user for a filename. However, `AskFile` returns only a string containing the name of a file. IDC's file-handling functions are detailed here:

`long fopen(string filename, string mode)`
Returns an integer file handle (or 0 on error) for use with all IDC file I/O functions. The `mode` parameter is similar to the modes used in C's `fopen` (`r` to read, `w` to write, and so on).

`void fclose(long handle)`
Closes the file specified by the file handle from `fopen`.

`long filelength(long handle)`
Returns the length of the indicated file or –1 on error.

`long fgetc(long handle)`
Reads a single byte from the given file. Returns –1 on error.

`long fputc(long val, long handle)`
Writes a single byte to the given file. Returns 0 on success or –1 on error.

`long fprintf(long handle, string format, ...)`
Writes a formatted string to the given file.

`long writestr(long handle, string str)`
Writes the specified string to the given file.

`string/long readstr(long handle)`
Reads a string from the given file. This function reads all characters (including non-ASCII) up to and including the next line feed (ASCII 0xA) character. Returns the string on success or –1 on end of file.

`long writelong(long handle, long val, long bigendian)`
Writes a 4-byte integer to the given file using big-endian (`bigendian` = 1) or little-endian (`bigendian` = 0) byte order.

`long readlong(long handle, long bigendian)`
Reads a 4-byte integer from the given file using big-endian (`bigendian` = 1) or little-endian (`bigendian` = 0) byte order.

`long writeshort(long handle, long val, long bigendian)`
Writes a 2-byte integer to the given file using big-endian (`bigendian` = 1) or little-endian (`bigendian` = 0) byte order.

`long readshort(long handle, long bigendian)`
Reads a 2-byte integer from the given file using big-endian (`bigendian` = 1) or little-endian (`bigendian` = 0) byte order.

`bool loadfile(long handle, long pos, long addr, long length)`
Reads `length` number of bytes from position `pos` in the given file and writes those bytes into the database beginning at address `addr`.

`bool savefile(long handle, long pos, long addr, long length)`
Writes `length` number of bytes beginning at database address `addr` to position `pos` in the given file.

Manipulating Database Names

The need to manipulate named locations arises fairly often in scripts. The following IDC functions are available for working with named locations in an IDA database:

`string Name(long addr)`
Returns the name associated with the given address or returns the empty string if the location has no name. This function does not return user-assigned names when the names are marked as local.

`string NameEx(long from, long addr)`
Returns the name associated with `addr`. Returns the empty string if the location has no name. This function returns user-defined local names if `from` is any address within a function that also contains `addr`.

`bool MakeNameEx(long addr, string name, long flags)`
Assigns the given name to the given address. The name is created with attributes specified in the `flags` bitmask. These flags are described in the help file documentation for `MakeNameEx` and are used to specify attributes such as whether the name is local or public or whether it should be listed in the names window.

`long LocByName(string name)`
Returns the address of the location with the given name. Returns BADADDR (–1) if no such name exists in the database.

`long LocByNameEx(long funcaddr, string localname)`
Searches for the given local name within the function containing `funcaddr`. Returns BADADDR (–1) if no such name exists in the given function.

Functions Dealing with Functions

Many scripts are designed to perform analysis of functions within a database. IDA assigns disassembled functions a number of attributes, such as the size of the function's local variable area or the size of the function's arguments on the runtime stack. The following IDC functions can be used to access information about functions within a database.

`long GetFunctionAttr(long addr, long attrib)`
Returns the requested attribute for the function containing the given address. Refer to the IDC help documentation for a list of attribute constants. As an example, to find the ending address of a function, use `GetFunctionAttr(addr, FUNCATTR_END);`.

`string GetFunctionName(long addr)`
Returns the name of the function that contains the given address or an empty string if the given address does not belong to a function.

`long NextFunction(long addr)`
Returns the starting address of the next function following the given address. Returns –1 if there are no more functions in the database.

long PrevFunction(long addr)
Returns the starting address of the nearest function that precedes the given address. Returns –1 if no function precedes the given address.

Use the `LocByName` function to find the starting address of a function given the function's name.

Code Cross-Reference Functions

Cross-references were covered in Chapter 9. IDC offers functions for accessing cross-reference information associated with any instruction. Deciding which functions meet the needs of your scripts can be a bit confusing. It requires you to understand whether you are interested in following the flows leaving a given address or whether you are interested in iterating over all of the locations that refer to a given address. Functions for performing both of the preceding operations are described here. Several of these functions are designed to support iteration over a set of cross-references. Such functions support the notion of a sequence of cross-references and require a `current` cross-reference in order to return a `next` cross-reference. Examples of using cross-reference iterators are provided in "Enumerating Cross-References" on page 272.

long Rfirst(long from)
Returns the first location to which the given address transfers control. Returns BADADDR (–1) if the given address refers to no other address.

long Rnext(long from, long current)
Returns the next location to which the given address (`from`) transfers control, given that `current` has already been returned by a previous call to `Rfirst` or `Rnext`. Returns BADADDR if no more cross-references exist.

long XrefType()
Returns a constant indicating the type of the last cross-reference returned by a cross-reference lookup function such as `Rfirst`. For code cross-references, these constants are `fl_CN` (near call), `fl_CF` (far call), `fl_JN` (near jump), `fl_JF` (far jump), and `fl_F` (ordinary sequential flow).

long RfirstB(long to)
Returns the first location that transfers control to the given address. Returns BADADDR (–1) if there are no references to the given address.

long RnextB(long to, long current)
Returns the next location that transfers control to the given address (`to`), given that `current` has already been returned by a previous call to `RfirstB` or `RnextB`. Returns BADADDR if no more cross-references to the given location exist.

Each time a cross-reference function is called, an internal IDC state variable is set that indicates the type of the last cross-reference that was returned. If you need to know what type of cross-reference you have received, then you must call `XrefType` prior to calling another cross-reference lookup function.

Data Cross-Reference Functions

The functions for accessing data cross-reference information are very similar to the functions used to access code cross-reference information. These functions are described here:

`long Dfirst(long from)`
: Returns the first location to which the given address refers to a data value. Returns BADADDR (–1) if the given address refers to no other addresses.

`long Dnext(long from, long current)`
: Returns the next location to which the given address (`from`) refers a data value, given that `current` has already been returned by a previous call to `Dfirst` or `Dnext`. Returns BADADDR if no more cross-references exist.

`long XrefType()`
: Returns a constant indicating the type of the last cross-reference returned by a cross-reference lookup function such as `Dfirst`. For data cross-references, these constants include `dr_O` (offset taken), `dr_W` (data write), and `dr_R` (data read).

`long DfirstB(long to)`
: Returns the first location that refers to the given address as data. Returns BADADDR (–1) if there are no references to the given address.

`long DnextB(long to, long current)`
: Returns the next location that refers to the given address (`to`) as data, given that `current` has already been returned by a previous call to `DfirstB` or `DnextB`. Returns BADADDR if no more cross-references to the given location exist.

As with code cross-references, if you need to know what type of cross-reference you have received, then you must call `XrefType` prior to calling another cross-reference lookup function.

Database Manipulation Functions

A number of functions exist for formatting the contents of a database. Here are descriptions of a few of these functions:

`void MakeUnkn(long addr, long flags)`
: Undefines the item at the specified address. The flags (see the IDC documentation for `MakeUnkn`) dictate whether subsequent items will also be undefined and whether any names associated with undefined items will be deleted. Related function `MakeUnknown` allows you to undefine large blocks of data.

`long MakeCode(long addr)`
: Converts the bytes at the specified address into an instruction. Returns the length of the instruction or 0 if the operation fails.

bool MakeByte(long addr)
Converts the item at the specified address into a data byte. MakeWord and MakeDword are also available.

bool MakeComm(long addr, string comment)
Adds a regular comment at the given address.

bool MakeFunction(long begin, long end)
Converts the range of instructions from begin to end into a function. If end is specified as BADADDR (-1), IDA attempts to automatically identify the end of the function by locating the function's return instruction.

bool MakeStr(long begin, long end)
Creates a string of the current string type (as returned by GetStringType), spanning the bytes from begin to end - 1. If end is specified as BADADDR, IDA attempts to automatically identify the end of the string.

Many other Make*XXX* functions exist that offer behavior similar to the functions just described. Please refer to the IDC documentation for a full list of these functions.

Database Search Functions

The majority of IDA's search capabilities are accessible in IDC in the form of various Find*XXX* functions, some of which are described here. The flags parameter used in the Find*XXX* functions is a bitmask that specifies the behavior of the find operation. Three of the more useful flags are SEARCH_DOWN, which causes the search to scan toward higher addresses; SEARCH_NEXT, which skips the current occurrence in order to search for the next occurrence; and SEARCH_CASE, which causes binary and text searches to be performed in a case-sensitive manner.

long FindCode(long addr, long flags)
Searches for an instruction from the given address.

long FindData(long addr, long flags)
Searches for a data item from the given address.

long FindBinary(long addr, long flags, string binary)
Searches for a sequence of bytes from the given address. The binary string specifies a sequence of hexadecimal byte values. If SEARCH_CASE is not specified and a byte value specifies an uppercase or lowercase ASCII letter, then the search will also match corresponding, complementary case values. For example, "41 42" will match "61 62" (and "61 42") unless the SEARCH_CASE flag is set.

long FindText(long addr, long flags, long row, long column, string text)
Searches for a text string from the given column on the given line (row) at the given address. Note that the disassembly text at a given address may span several lines, hence the need to specify on which line the search should begin.

Also note that SEARCH_NEXT does not define the direction of search, which may be either up or down according to the SEARCH_DOWN flag. In addition, when SEARCH_NEXT is not specified, it is perfectly reasonable for a Find*XXX* function to return the same address that was passed in as the addr argument when the item at addr satisfies the search.

Disassembly Line Components

From time to time it is useful to extract the text, or portions of the text, of individual lines in a disassembly listing. The following functions provide access to various components of a disassembly line:

string GetDisasm(long addr)
Returns disassembly text for the given address. The returned text includes any comments but does not include address information.

string GetMnem(long addr)
Returns the mnemonic portion of the instruction at the given address.

string GetOpnd(long addr, long opnum)
Returns the text representation of the specified operand at the specified address. Operands are numbered from zero beginning with the leftmost operand.

long GetOpType(long addr, long opnum)
Returns an integer representing the type for the given operand at the given address. Refer to the IDC documentation for GetOpType for a complete list of operand type codes.

long GetOperandValue(long addr, long opnum)
Returns the integer value associated with the given operand at the given address. The nature of the returned value depends on the type of the given operand as specified by GetOpType.

string CommentEx(long addr, long type)
Returns the text of any comment present at the given address. If type is 0, the text of the regular comment is returned. If type is 1, the text of the repeatable comment is returned. If no comment is present at the given address, an empty string is returned.

IDC Scripting Examples

At this point it is probably useful to see some examples of scripts that perform specific tasks. For the remainder of the chapter we present some fairly common situations in which a script can be used to answer a question about a database.

Enumerating Functions

Many scripts operate on individual functions. Examples include generating the call tree rooted at a specific function, generating the control flow graph of a function, or analyzing the stack frames of every function in a database.

Listing 15-1 iterates through every function in a database and prints basic information about each function, including the start and end addresses of the function, the size of the function's arguments, and the size of the function's local variables. All output is sent to the output window.

```
#include <idc.idc>
static main() {
   auto addr, end, args, locals, frame, firstArg, name, ret;
   addr = 0;
   for (addr = NextFunction(addr); addr != BADADDR; addr = NextFunction(addr)) {
      name = Name(addr);
      end = GetFunctionAttr(addr, FUNCATTR_END);
      locals = GetFunctionAttr(addr, FUNCATTR_FRSIZE);
      frame = GetFrame(addr);     // retrieve a handle to the function's stack frame
      ret = GetMemberOffset(frame, " r");  // " r" is the name of the return address
      if (ret == -1) continue;
      firstArg = ret + 4;
      args = GetStrucSize(frame) - firstArg;
      Message("Function: %s, starts at %x, ends at %x\n", name, addr, end);
      Message("   Local variable area is %d bytes\n", locals);
      Message("   Arguments occupy %d bytes (%d args)\n", args, args / 4);
   }
}
```

Listing 15-1: Function enumeration script

This script uses some of IDC's structure-manipulation functions to obtain a handle to each function's stack frame (`GetFrame`), determine the size of the stack frame (`GetStrucSize`), and determine the offset of the saved return address within the frame (`GetMemberOffset`). The first argument to the function lies 4 bytes beyond the saved return address. The size of the function's argument area is computed as the space between the first argument and the end of the stack frame. Since IDA can't generate stack frames for imported functions, this script tests whether the function's stack frame contains a saved return address as a simple means of identifying calls to an imported function.

Enumerating Instructions

Within a given function, you may want to enumerate every instruction. Listing 15-2 counts the number of instructions contained in the function identified by the current cursor position:

```
#include <idc.idc>
static main() {
   auto func, end, count, inst;
❶  func = GetFunctionAttr(ScreenEA(), FUNCATTR_START);
   if (func != -1) {
❷     end = GetFunctionAttr(func, FUNCATTR_END);
      count = 0;
      inst = func;
      while (inst < end) {
```

```
            count++;
❸           inst = FindCode(inst, SEARCH_DOWN | SEARCH_NEXT);
         }
         Warning("%s contains %d instructions\n", Name(func), count);
      }
      else {
         Warning("No function found at location %x", ScreenEA());
      }
   }
```

Listing 15-2: Instruction enumeration script

The function begins ❶ by using `GetFunctionAttr` to determine the start address of the function containing the cursor address (`ScreenEA()`). If the beginning of a function is found, the next step ❷ is to determine the end address for the function, once again using the `GetFunctionAttr` function. Once the function has been bounded, a loop is executed to step through successive instructions in the function by using the search functionality of the `FindCode` function ❸. In this example, the `Warning` function is used to display results, since only a single line of output will be generated by the function and output displayed in a Warning dialog is much more obvious than output generated in the message window. Note that this example assumes that all of the instructions within the given function are contiguous. An alternative approach might replace the use of `FindCode` with logic to iterate over all of the code cross-references for each instruction within the function. Properly written, this second approach would handle noncontiguous, also known as "chunked," functions.

Enumerating Cross-References

Iterating through cross-references can be confusing because of the number of functions available for accessing cross-reference data and the fact that code cross-references are bidirectional. In order to get the data you want, you need to make sure you are accessing the proper type of cross-reference for your situation. In our first cross-reference example, shown in Listing 15-3, we derive the list of all function calls made within a function by iterating through each instruction in the function to determine if the instruction calls another function. One method of doing this might be to parse the results of `GetMnem` to look for `call` instructions. This would not be a very portable solution, because the instruction used to call a function varies among CPU types. Second, additional parsing would be required to determine exactly which function was being called. Cross-references avoid each of these difficulties because they are CPU-independent and directly inform us about the target of the cross-reference.

```
#include <idc.idc>
static main() {
  auto func, end, target, inst, name, flags, xref;
  flags = SEARCH_DOWN | SEARCH_NEXT;
  func = GetFunctionAttr(ScreenEA(), FUNCATTR_START);
```

```
  if (func != -1) {
    name = Name(func);
    end = GetFunctionAttr(func, FUNCATTR_END);
    for (inst = func; inst < end; inst = FindCode(inst, flags)) {
      for (target = Rfirst(inst); target != BADADDR; target = Rnext(inst, target)) {
        xref = XrefType();
        if (xref == fl_CN || xref == fl_CF) {
          Message("%s calls %s from 0x%x\n", name, Name(target), inst);
        }
      }
    }
  }
  else {
    Warning("No function found at location %x", ScreenEA());
  }
}
```

Listing 15-3: Enumerating function calls

In this example, we must iterate through each instruction in the function. For each instruction, we must then iterate through each cross-reference from the instruction. We are interested only in cross-references that call other functions, so we must test the return value of `XrefType` looking for `fl_CN` or `fl_CF`-type cross-references. Here again, this particular solution handles only functions whose instructions happen to be contiguous. Given that the script is already iterating over the cross-references from each instruction, it would not take many changes to produce a flow-driven analysis instead of the address-driven analysis seen here.

Another use for cross-references is to determine every location that references a particular location. For example, if we wanted to create a low-budget security analyzer, we might be interested in highlighting all calls to functions such as `strcpy` and `sprintf`.

DANGEROUS FUNCTIONS

The C functions `strcpy` and `sprintf` are generally acknowledged as dangerous to use because they allow for unbounded copying into destination buffers. While each may be safely used by programmers who conduct proper checks on the size of source and destination buffers, such checks are all too often forgotten by programmers unaware of the dangers of these functions. The `strcpy` function, for example, is declared as follows:

```
char *strcpy(char *dest, const char *source);
```

The `strcpy` function's defined behavior is to copy all characters up to and including the first null termination character encountered in the source buffer to the given destination buffer (`dest`). The fundamental problem is that there is no way to determine, at runtime, the size of any array. In this instance, `strcpy` has no means to determine whether the capacity of the destination buffer is sufficient to hold all of the data to be copied from source. Such unchecked copy operations are a major cause of buffer overflow vulnerabilities.

In the example shown in Listing 15-4, we work in reverse to iterate across all of the cross-references *to* (as opposed to *from* in the preceding example) a particular symbol:

```
#include <idc.idc>
static list_callers(bad_func) {
   auto func, addr, xref, source;
❶  func = LocByName(bad_func);
   if (func == BADADDR) {
      Warning("Sorry, %s not found in database", bad_func);
   }
   else {
❷     for (addr = RfirstB(func); addr != BADADDR; addr = RnextB(func, addr)) {
❸       xref = XrefType();
❹       if (xref == fl_CN || xref == fl_CF) {
❺           source = GetFunctionName(addr);
❻           Message("%s is called from 0x%x in %s\n", bad_func, addr, source);
         }
      }
   }
}
static main() {
   list_callers("_strcpy");
   list_callers("_sprintf");
}
```

Listing 15-4: Enumerating a function's callers

In this example, the `LocByName` ❶ function is used to find the address of a given (by name) bad function. If the function's address is found, a loop ❷ is executed in order to process all cross-references to the bad function. For each cross-reference, if the cross-reference type ❸ is determined to be a call-type ❹ cross-reference, the calling function's name is determined ❺ and is displayed to the user ❻.

It is important to note that some modifications may be required to perform a proper lookup of the name of an imported function. In ELF executables in particular, which combine a procedure linkage table (PLT) with a global offset table (GOT) to handle the details of linking to shared libraries, the names that IDA assigns to imported functions may be less than clear. For example, a PLT entry may appear to be named `_memcpy`, when in fact it is named `.memcpy` and IDA has replaced the dot with an underscore because IDA considers dots invalid characters within names. Further complicating matters is the fact that IDA may actually create a symbol named `memcpy` that resides in a section that IDA names `extern`. When attempting to enumerate cross-references to `memcpy`, we are interested in the PLT version of the symbol because this is the version that is called from other functions in the program and thus the version to which all cross-references would refer.

Enumerating Exported Functions

In Chapter 13 we discussed the use of `idsutils` to generate *.ids* files that describe the contents of shared libraries. Recall that the first step in generating a *.ids* file involves generating a *.idt* file, which is a text file containing descriptions of each exported function contained in the library. IDC contains functions for iterating through the functions that are exported by a shared library. The script shown in Listing 15-5 can be run to generate an *.idt* file after opening a shared library with IDA:

```
#include <idc.idc>
static main() {
   auto entryPoints, i, ord, addr, name, purged, file, fd;
   file = AskFile(1, "*.idt", "Select IDT save file");
   fd = fopen(file, "w");
   entryPoints = GetEntryPointQty();
   fprintf(fd, "ALIGNMENT 4\n");
   fprintf(fd, "0 Name=%s\n", GetInputFile());
   for (i = 0; i < entryPoints; i++) {
      ord = GetEntryOrdinal(i);
      if (ord == 0) continue;
      addr = GetEntryPoint(ord);
      if (ord == addr) {
         continue; //entry point has no ordinal
      }
      name = Name(addr);
      fprintf(fd, "%d Name=%s", ord, name);
      purged = GetFunctionAttr(addr, FUNCATTR_ARGSIZE);
      if (purged > 0) {
         fprintf(fd, " Pascal=%d", purged);
      }
      fprintf(fd, "\n");
   }
}
```

Listing 15-5: A script to generate .idt *files*

The output of the script is saved to a file chosen by the user. New functions introduced in this script include `GetEntryPointQty`, which returns the number of symbols exported by the library; `GetEntryOrdinal`, which returns an ordinal number (an index into the library's export table); `GetEntryPoint`, which returns the address associated with an exported function that has been identified by ordinal number; and `GetInputFile`, which returns the name of the file that was loaded into IDA.

Finding and Labeling Function Arguments

Versions of GCC later than 3.4 use `mov` statements rather than `push` statements in x86 binaries to place function arguments into the stack before calling a function. Occasionally this causes some analysis problems for IDA (newer versions of IDA handle this situation better), because the analysis engine

relies on finding `push` statements to pinpoint locations at which arguments are pushed for a function call. The following listing shows an IDA disassembly when parameters are pushed onto the stack:

```
.text:08048894                push    0               ; protocol
.text:08048896                push    1               ; type
.text:08048898                push    2               ; domain
.text:0804889A                call    _socket
```

Note the comments that IDA has placed in the right margin. Such commenting is possible only when IDA recognizes that parameters are being pushed and when IDA knows the signature of the function being called. When `mov` statements are used to place parameters onto the stack, the resulting disassembly is somewhat less informative, as shown here:

```
.text:080487AD                mov     [esp+8], 0
.text:080487B5                mov     [esp+4], 1
.text:080487BD                mov     [esp], 2
.text:080487C4                call    _socket
```

In this case, IDA has failed to recognize that the three `mov` statements preceding the call are being used to set up the parameters for the function call. As a result, we get less assistance from IDA in the form of automatic comments in the disassembly.

Here we have a situation where a script might be able to restore some of the information that we are accustomed to seeing in our disassemblies. Listing 15-6 is a first effort at automatically recognizing instructions that are setting up parameters for function calls:

```
#include <idc.idc>
static main() {
  auto addr, op, end, idx;
  auto func_flags, type, val, search;
  search = SEARCH_DOWN | SEARCH_NEXT;
  addr = GetFunctionAttr(ScreenEA(), FUNCATTR_START);
  func_flags = GetFunctionFlags(addr);
  if (func_flags & FUNC_FRAME) {  //Is this an ebp-based frame?
    end = GetFunctionAttr(addr, FUNCATTR_END);
    for (; addr < end && addr != BADADDR; addr = FindCode(addr, search)) {
      type = GetOpType(addr, 0);
      if (type == 3) {  //Is this a register indirect operand?
        if (GetOperandValue(addr, 0) == 4) {   //Is the register esp?
          MakeComm(addr, "arg_0");  //[esp] equates to arg_0
        }
      }
```

```
      else if (type == 4) {  //Is this a register + displacement operand?
        idx = strstr(GetOpnd(addr, 0), "[esp"); //Is the register esp?
        if (idx != -1) {
          val = GetOperandValue(addr, 0);   //get the displacement
          MakeComm(addr, form("arg_%d", val));  //add a comment
        }
      }
    }
  }
}
```

Listing 15-6: Automating parameter recognition

The script works only on EBP-based frames and relies on the fact that when parameters are moved into the stack prior to a function call, GCC generates memory references relative to `esp`. The script iterates through all instructions in a function; for each instruction that writes to a memory location using `esp` as a base register, the script determines the depth within the stack and adds a comment indicating which parameter is being moved. The `GetFunctionFlags` function offers access to various flags associated with a function, such as whether the function uses an EBP-based stack frame. Running the script in Listing 15-6 yields the annotated disassembly shown here:

```
.text:080487AD                mov     [esp+8], 0   ; arg_8
.text:080487B5                mov     [esp+4], 1   ; arg_4
.text:080487BD                mov     [esp], 2    ; arg_0
.text:080487C4                call    _socket
```

The comments aren't particularly informative. However, we can now tell at a glance that the three `mov` statements are used to place parameters onto the stack, which is a step in the right direction. By extending the script a bit further and exploring some more of IDC's capabilities, we can come up with a script that provides almost as much information as IDA does when it properly recognizes parameters. The output of the final product is shown here:

```
.text:080487AD                mov     [esp+8], 0   ;  int protocol
.text:080487B5                mov     [esp+4], 1   ;  int type
.text:080487BD                mov     [esp], 2    ;  int domain
.text:080487C4                call    _socket
```

The extended version of the script in Listing 15-6, which is capable of incorporating data from function signatures into comments, is available on this book's website.[8]

8. See *http://www.idabook.com/ch15_examples*.

Emulating Assembly Language Behavior

There are a number of reasons why you might need to write a script that emulates the behavior of a program you are analyzing. For example, the program you are studying may be self-modifying, as many malware programs are, or the program may contain some encoded data that gets decoded when it is needed at runtime. Without running the program and pulling the modified data out of the running process's memory, how can you understand the behavior of the program? The answer may lie with an IDC script. If the decoding process is not terribly complex, you may be able to quickly write an IDC script that performs the same actions that are performed by the program when it runs. Using a script to decode data in this way eliminates the need to run a program when you don't know what the program does or you don't have access to a platform on which you can run the program. An example of the latter case might occur if you were examining a MIPS binary with your Windows version of IDA. Without any MIPS hardware, you would not be able to execute the MIPS binary and observe any data decoding it might perform. You could, however, write an IDC script to mimic the behavior of the binary and make the required changes within the IDA database, all with no need for a MIPS execution environment.

The following x86 code was extracted from a DEFCON[9] Capture the Flag binary.[10]

```
.text:08049EDE                  mov     [ebp+var_4], 0
.text:08049EE5
.text:08049EE5 loc_8049EE5:
.text:08049EE5                  cmp     [ebp+var_4], 3C1h
.text:08049EEC                  ja      short locret_8049F0D
.text:08049EEE                  mov     edx, [ebp+var_4]
.text:08049EF1                  add     edx, 804B880h
.text:08049EF7                  mov     eax, [ebp+var_4]
.text:08049EFA                  add     eax, 804B880h
.text:08049EFF                  mov     al, [eax]
.text:08049F01                  xor     eax, 4Bh
.text:08049F04                  mov     [edx], al
.text:08049F06                  lea     eax, [ebp+var_4]
.text:08049F09                  inc     dword ptr [eax]
.text:08049F0B                  jmp     short loc_8049EE5
```

This code decodes a private key that has been embedded within the program binary. Using the IDC script shown in Listing 15-7, we can extract the private key without running the program:

```
auto var_4, edx, eax, al;
var_4 = 0;
while (var_4 <= 0x3C1) {
   edx = var_4;
```

9. See *http://www.defcon.org/*.

10. Courtesy of Kenshoto, the organizers of CTF at DEFCON 15. Capture the Flag is an annual hacking competition held at DEFCON.

```
    edx = edx + 0x804B880;
    eax = var_4;
    eax = eax + 0x804B880;
    al = Byte(eax);
    al = al ^ 0x4B;
    PatchByte(edx, al);
    var_4++;
}
```

Listing 15-7: Emulating assembly language with IDC

Listing 15-7 is a fairly literal translation of the preceding assembly language sequence generated according to the following rather mechanical rules.

1. For each stack variable and register used in the assembly code, declare an IDC variable.
2. For each assembly language statement, write an IDC statement that mimics its behavior.
3. Reading and writing stack variables is emulated by reading and writing the corresponding variable declared in your IDC script.
4. Reading from a nonstack location is accomplished using the `Byte`, `Word`, or `Dword` function, depending on the amount of data being read (1, 2, or 4 bytes).
5. Writing to a nonstack location is accomplished using the `PatchByte`, `PatchWord`, or `PatchDword` function, depending on the amount of data being written.
6. In general, if the code appears to contain a loop for which the termination condition is not immediately obvious, it is easiest to begin with an infinite loop such as `while (1) {}` and then insert a `break` statement when you encounter statements that cause the loop to terminate.
7. When the assembly code calls functions, things get complicated. In order to properly simulate the behavior of the assembly code, you must find a way to mimic the behavior of the function that has been called, including providing a return value that makes sense within the context of the code being simulated. This fact alone may preclude the use of IDC as a tool for emulating the behavior of an assembly language sequence.

The important thing to understand when developing scripts such as the previous one is that it is not absolutely necessary to fully understand how the code you are emulating behaves on a global scale. It is often sufficient to understand only one or two instructions at a time and generate correct IDC translations for those instructions. If each instruction has been correctly translated into IDC, then the script as a whole should properly mimic the complete functionality of the original assembly code. We can delay further study of the assembly language algorithm until after the IDC script has been completed, at which point we can use the IDC script to enhance our

understanding of the underlying assembly. Once we spend some time considering how our example algorithm works, we might shorten the preceding IDC script to the following:

```
auto var_4, addr;
for (var_4 = 0; var_4 <= 0x3C1; var_4++) {
   addr = 0x804B880 + var_4;
   PatchByte(addr, Byte(addr) ^ 0x4B);
}
```

As an alternative, if we did not wish to modify the database in any way, we could replace the `PatchByte` function with a call to `Message` if we were dealing with ASCII data, or as an alternative we could write the data to a file if we were dealing with binary data.

IDAPython

IDAPython is a plug-in developed by Gergely Erdelyi that integrates a Python interpreter into IDA. Combined with supplied Python bindings, this plug-in allows you to write Python scripts with full access to all of the capabilities of the IDC scripting language. One clear advantage gained with IDAPython is access to Python's native data-handling capabilities as well as the full range of Python modules. In addition, IDAPython exposes a significant portion of IDA's SDK functionality, allowing for far more powerful scripting than is possible using IDC. IDAPython has developed quite a following in the IDA community. Ilfak's blog[11] contains numerous interesting examples of problem solving with Python scripts, while questions, answers, and many other useful IDAPython scripts are frequently posted in the forums at OpenRCE.org.[12] In addition, third-party tools such as BinNavi[13] from Zynamics rely on IDA and IDAPython in order to perform various subtasks required by the tools.

Since IDA 5.4, Hex-Rays has been including IDAPython as a standard plug-in. Source code for the plug-in is available for download on the IDAPython project page,[14] and API documentation is available on the Hex-Rays website.[15] IDA enables the plug-in only when Python is found to be installed on the computer on which you are running IDA. The Windows version of IDA ships with and installs a compatible version of Python,[16] while the Linux and OS X versions of IDA leave proper installation of Python up to you. On Linux, the current version of IDA (6.1) looks for Python 2.6. IDAPython is compatible with Python 2.7, and IDA will work just fine if you create symlinks

11. See *http://www.hexblog.com*.

12. See *http://www.openrce.org/articles/*.

13. See *http://www.zynamics.com/binnavi.html*.

14. See *http://code.google.com/p/idapython/*.

15. See *http://www.hex-rays.com/idapro/idapython_docs/index.html*.

16. See *http://www.python.org/*.

from the required Python 2.6 libraries to your existing Python 2.7 libraries. If you have Python 2.7, a command similar to the following will create the symlink that will make IDA happy:

```
# ln -s /usr/lib/libpython2.7.so.1.0 /usr/lib/libpython2.6.so.1
```

OS X users may find that the version of Python that ships with OS X is older than that required by IDA. If this is the case, a suitable Python installer should be downloaded from *www.python.org*.[17]

Using IDAPython

IDAPython bridges Python code into IDA by making available three Python modules, each serving a specific purpose. Access to the core IDA API (as exposed via the SDK) is made available with the `idaapi` module. All of the functions present in IDC are made available in IDAPython's `idc` module. The third module that ships with IDAPython is `idautils`, which provides a number of utility functions, many of which yield Python lists of various database-related objects such as functions or cross-references. Modules `idc` and `idautils` are automatically imported for all IDAPython scripts. If you need `idaapi`, on the other hand, you must import it yourself.

When using IDAPython, keep in mind that the plug-in embeds a single instance of the Python interpreter into IDA. This interpreter is not destroyed until you close IDA. As a result, you can view all of your scripts and statements as if they are running within a single Python shell session. For example, once you have imported the `idaapi` module for the first time in your IDA session, you need never import it again until you restart IDA. Similarly, initialized variables and function definitions retain their values until they are redefined or until you quit IDA.

There are a number of strategies for learning IDA's Python API. If you already have some experience using IDC or programming with the IDA SDK, then you should feel right at home with the `idaapi` and `idc` modules. A quick review of the additional features in the `idautils` module should be all you really need to start making full use of IDAPython. If you have prior experience with IDC or the SDK, then you might dive into the Hex-Ray's documentation for the Python API to develop a feel for the capabilities it offers. Remember that the `idc` module basically mirrors the IDC API and that you may find the list of IDC functions in IDA's built-in help to be quite useful. Similarly, the descriptions of IDC functions presented earlier in this chapter are equally applicable to the corresponding functions in the `idc` module.

17. See *http://www.python.org/download/mac/*.

IDAPython Scripting Examples

By way of offering a compare and contrast between IDC and IDAPython, the following sections present the same example cases seen previously in the discussion of IDC. Wherever possible we endeavor to make maximum use of Python-specific features to demonstrate some of the efficiencies that can be gained by scripting in Python.

Enumerating Functions

One of the strengths of IDAPython is the way that it uses Python's powerful datatypes to simplify access to collections of database objects. In Listing 15-8, we reimplement the function enumeration script of Listing 15-1 in Python. Recall that the purpose of this script is to iterate over every function in a database and print basic information about each function, including the start and end addresses of the function, the size of the function's arguments, and the size of the function's local variable space. All output is sent to the output window.

```
funcs = Functions()❶
for f in funcs:❷
   name = Name(f)
   end = GetFunctionAttr(f, FUNCATTR_END)
   locals = GetFunctionAttr(f, FUNCATTR_FRSIZE)
   frame = GetFrame(f)     # retrieve a handle to the function's stack frame
   if frame is None: continue
   ret = GetMemberOffset(frame, " r")  # " r" is the name of the return address
   if ret == -1: continue
   firstArg = ret + 4
   args = GetStrucSize(frame) - firstArg
   Message("Function: %s, starts at %x, ends at %x\n" % (name, f, end))
   Message("   Local variable area is %d bytes\n" % locals)
   Message("   Arguments occupy %d bytes (%d args)\n" % (args, args / 4))
```

Listing 15-8: Function enumeration using Python

For this particular script, the use of Python gains us little in the way of efficiency other than the use of the `Functions` ❶ list generator, which facilitates the `for` loop at ❷.

Enumerating Instructions

Listing 15-9 demonstrates how the instruction-counting script of Listing 15-2 might be written in Python, taking advantage of the list generators available in the `idautils` module.

```
from idaapi import *
func = get_func(here())❶  # here() is synonymous with ScreenEA()
if not func is None:
   fname = Name(func.startEA)
   count = 0
```

```
   for i in FuncItems(func.startEA)❷: count = count + 1
   Warning("%s contains %d instructions\n" % (fname,count))
else:
   Warning("No function found at location %x" % here())
```

Listing 15-9: Instruction enumeration in Python

Differences from the IDC version include the use of an SDK function ❶ (accessed via `idaapi`) to retrieve a reference to a function object (specifically a `func_t`) and the use of the `FuncItems` generator ❷ (from `idautils`) to provide easy iteration over all of the instructions within the function. Because we can't use Python's `len` function on a generator, we are still obligated to step through the generator list in order to count each instruction one at a time.

Enumerating Cross-References

The `idautils` module contains several generator functions that build cross-reference lists in a somewhat more intuitive way than we saw in IDC. Listing 15-10 rewrites the function call enumeration script that we saw previously in Listing 15-3.

```
from idaapi import *
func = get_func(here())
if not func is None:
   fname = Name(func.startEA)
   items = FuncItems(func.startEA)
   for i in items:
      for xref in XrefsFrom(i, 0):❶
         if xref.type == fl_CN or xref.type == fl_CF:
            Message("%s calls %s from 0x%x\n" % (fname, Name(xref.to), i))
else:
   Warning("No function found at location %x" % here())
```

Listing 15-10: Enumerating function calls using Python

New in this script is the use of the `XrefsFrom` generator ❶ (from `idautils`) to step through all cross-references from the current instruction. `XrefsFrom` returns a reference to an `xrefblk_t` object that contains detailed information about the current cross-reference.

Enumerating Exported Functions

Listing 15-11 is the Python version of the *.idt* generator script from Listing 15-5.

```
file = AskFile(1, "*.idt", "Select IDT save file")
with open(file, 'w') as fd:
   fd.write("ALIGNMENT 4\n")
   fd.write("0 Name=%s\n" % GetInputFile())
   for i in range(GetEntryPointQty()):
      ord = GetEntryOrdinal(i)
      if ord == 0: continue
      addr = GetEntryPoint(ord)
```

```
if ord == addr: continue   #entry point has no ordinal
fd.write("%d Name=%s" % (ord, Name(addr)))
purged = GetFunctionAttr(addr, FUNCATTR_ARGSIZE)
if purged > 0:
   fd.write(" Pascal=%d" % purged)
fd.write("\n")
```

Listing 15-11: A Python script to generate IDT files

The two scripts look remarkably similar because IDAPython has no generator function for entry-point lists, so we are left to use the same set of functions that were used in Listing 15-5. One difference worth noting is that IDAPython deprecates IDC's file-handling functions in favor of Python's built-in file-handling functions.

Summary

Scripting provides a powerful means for extending IDA's capabilities. Through the years, scripts have been used in a number of innovative ways to fill the needs of IDA users. Many useful scripts are available for download on the Hex-Rays website as well as the mirror site of the former IDA Palace.[18] IDA scripts are perfect for small tasks and rapid development, but they are not ideally suited for all situations.

One of the principal limitations of the IDC language is its lack of support for complex datatypes and the lack of access to a more fully featured API such as the C standard library or the Windows API. At the expense of greater complexity, we can lift these limitations by moving away from scripted extensions and toward compiled extensions. As we will show in the next chapter, compiled extensions require the use of the IDA software development kit (SDK), which has a steeper learning curve than either IDC or IDAPython. However, the power available when developing extensions with the SDK is usually well worth the effort spent learning how to use it.

18. See *http://old.idapalace.net/*.